LAB3 OS

Name: Aasem Mohamed Henedy

ID: 5580

GROUP: 4

SEC: 2

sem\_t counterSemaphore;

sem\_t bufferSemaphore;

int counter = 0;

int buffer[b] = {-1};

int bufferFill = 0; //fill count in buffer

int buffer\_write\_pos = 0; //next writing position in buffer

int buffer\_read\_pos = 0; //next reading position in buffer

void\* mcounter(void\* id\_P) {

int id = (int)id\_P;

int semaphore\_val;

while(1) {

usleep((rand()%5)\*0.5\*1e6);

printf("Counter thread %d: received a message\n",id);

sem\_getvalue(&counterSemaphore, &semaphore\_val); //check if counter is currently used by another thread

if(semaphore\_val < 1) printf("Counter thread %d: waiting to write\n",id);

sem\_wait(&counterSemaphore);

counter++;

printf("Counter thread %d: now adding to counter, new value is %d\n",id,counter);

usleep((rand()%5)\*0.25\*1e6);

sem\_post(&counterSemaphore);

}

}

The general idea of semaphores is there must be 1 thread in the critical section that is the definition of mutual exclusion that semaphores apply.

In the upper function “mcounter” the counter thread receive a message and check the value of semaphore if smaller than 1 that means it is busy if no so it enters the critical section and increment counter value.

void\* mmonitor( ) {

int semaphore\_val;

int temp\_counter\_val;

bool buffer\_wait = false;

while(1) {

usleep((rand()%5)\*0.5\*1e6);

sem\_getvalue(&counterSemaphore, &semaphore\_val); //check if counter is currently used by another thread

if(semaphore\_val < 1) printf("Monitor thread: waiting to read counter\n");

if(!buffer\_wait){

sem\_wait(&counterSemaphore);

temp\_counter\_val = counter;

counter = 0;

printf("Monitor thread: reading a count value of %d\n",temp\_counter\_val);

usleep((rand()%5)\*0.25\*1e6);

sem\_post(&counterSemaphore);

}

sem\_getvalue(&bufferSemaphore, &semaphore\_val); //check if buffer is currently used by another thread

if(semaphore\_val < 1) printf("Monitor thread: waiting to write to buffer\n");

sem\_wait(&bufferSemaphore);

if(bufferFill == b) {

printf("Monitor thread: buffer full !\n");

buffer\_wait = true;

}

else {

buffer[buffer\_write\_pos] = temp\_counter\_val;

printf("Monitor thread: writing to buffer at position %d\n",buffer\_write\_pos+1);

if(buffer\_write\_pos == b-1) buffer\_write\_pos = 0;

else buffer\_write\_pos++;

bufferFill++;

buffer\_wait = false;

}

usleep((rand()%5)\*0.25\*1e6);

sem\_post(&bufferSemaphore);

}

}

In second function “mmonitor” we check value of counter if smaller than 1 that means it is busy and we wait for critical section to be free and when it is free we read from counter its value and we check buffer too if it is full or no in order to write in it.

void\* mmcollector() {

int semaphore\_val;

int temp\_read\_val;

while(1) {

usleep((rand()%5)\*0.5\*1e6);

sem\_getvalue(&bufferSemaphore, &semaphore\_val); //check if buffer is currently used by another thread

if(semaphore\_val < 1) printf("Collector thread: waiting to read from buffer\n");

sem\_wait(&bufferSemaphore);

if(bufferFill == 0) {

printf("Collector thread: nothing is in the buffer !\n");

}

else {

temp\_read\_val = buffer[buffer\_read\_pos];

printf("Collector thread: reading from buffer at position %d, read value is %d\n",buffer\_read\_pos+1,temp\_read\_val);

if(buffer\_read\_pos == b-1) buffer\_read\_pos = 0;

else buffer\_read\_pos++;

bufferFill--;

}

usleep((rand()%5)\*0.25\*1e6);

sem\_post(&bufferSemaphore);

}

}

We check if the buffer is currently used or no if no we check if it is empty or no in order to read from buffer the values in it.

int main() {

sem\_init(&counterSemaphore, 0, 1); //initialize semaphores to 1

sem\_init(&bufferSemaphore, 0, 1);

pthread\_t mcounter\_threads[N], mmonitor\_thread, mmcollector\_thread;

int ids[N];

for (int i = 0; i < N; i++) {

ids[i]=i+1;

pthread\_create(&mcounter\_threads[i], NULL, mcounter, &ids[i]);

}

pthread\_create(&mmonitor\_thread, NULL, mmonitor, NULL);

pthread\_create(&mmcollector\_thread, NULL, mmcollector, NULL);

sleep(10);

return 0;

pthread\_join(mmcollector\_thread, NULL);

pthread\_join(mmonitor\_thread, NULL);

for (int i = 0; i < N; i++) pthread\_join(mcounter\_threads[i], NULL);

return 0;

}

In the main function we use pthreadcreate to create threads for functions we used and pthread join to let the threads take actions after the one before it ends its work in the critical section.